# HPC presentation

Yoshitaka Sakurai (B4)

November 20, 2017

# Paper

DCatch: Automatically Detecting Distributed Concurrency Bugs in Cloud Systems

- Haopeng Liu(University of Chicago)
- Guangpu Li(University of Chicago)
- Jeffrey F. Lukman(University of Chicago)
- Jiaxin Li(University of Chicago)
- Shan Lu(University of Chicago)
- Haryadi S. Gunawi(University of Chicago)
- Chen Tian (Huawei R&D Center)

# Introduction

# Motivation

Distributed cloud software infrastructures have emerged as a dominant backbone for modern applications.
But, it is challenging to guarantee reliablity due to wide-spreading software bugs.

- DCbugs
  - distributed concurrency bugs
  - the most troublesome among all types of bugs in distributed system

DCbugs are triggered by untimely interaction among nodes.

# DCbugs in Hadoop

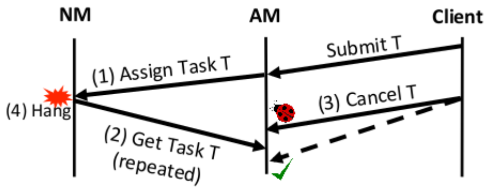


**Figure 1.** A Hadoop DCbug: Hang (buggy) if #3 happens before #2, or no failure (√) if the otherwise.

1. AM assign a task T to a container in NM
2. NM container tries to retrieve the content of task T from AM
3. T is canceled on the client's request
4. NM container hangs (waiting forever for AM to return task T)
   (T is already canceled in #3)

# DCbugs is difficult to avoid,detect and debug

DCbugs is difficult to avoid, detect and debug

- non-deterministic
- hide in the huge staet space of distributed system spreading across multiple nodes

**This paper present the first attempts in building DCbug detection tool for distributed systems.**

# Related work

Model checking

- Distributed system model checkers (dmck)
- dmck is powerful
- Dmck does not scale
    - The more events included, the larger the state spaceto be explored

Verification

- Strong solution (no false positive and negative)
- require thousands of lines of proof for every protocol

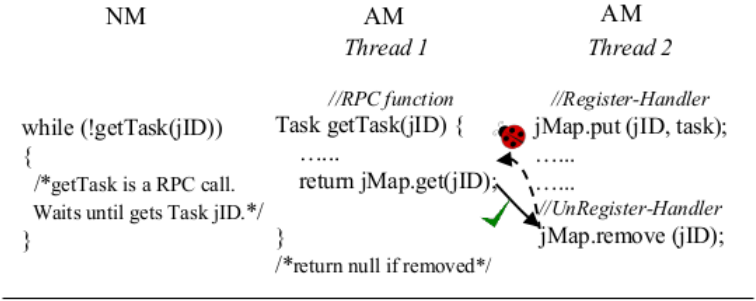
LCbug and DCbug detection

- LCbug is Local Concurrency bugs
- Many bug detectors for LCbug have been proposed

# Opportunities

DCbugs have fundamentally similar root causes as LCbugs

- Both are conflicting accesses to the same memory location



```
while (!getTask(jID))
{
  /*getTask is a RPC call.
  Waits until gets Task jID.*/
}
```

```
       //RPC function
Task getTask(jID) {
    ......
    return jMap.getTask(jID);

}
/*return null if removed*/
```

```
    //Register-Handler
jMap.put (jID, task);
......
......
//UnRegister-Handler
jMap.remove (jID);
```

**Figure 2.** Root cause of the DCbug shown in Fig. 1

DCbugs detection can re-use the theoretical foundation and work flow of LCbugs detection.

- abstract the causality relationship in distributed system into HB graph
- identify all pairs of concurrent conflicting memory access based on HB graph and treat them as DCbugs candidates.

HB graph(Happens-Before Graph) is discribed below

# Challenge

DCbugs is differ from LCbugs in several aspects

- More complicated timing relationship
  - concurrent accesses are conducted not only at thread level but also node level and event level
- Larger scales of system and bugs
  - Distributed system naturally run at a larger scale than single-machine
  - the larger bug scale also demands new techniques in bug impact analysis and bug exposing
- More subtle fault tolerance
  - Distributed systems contain inherent redundancy and aim to tolerate component failures.
  - So it is difficult to judge what are truly harmful bugs

# DCatch

This paper present DCatch
DCatch is a pilot solution in the world of DCbug detection The

design of the DCatch contains two stage

1. design HB model for distributed system
2. design DCatch tool components

# DCatch Happens-Before(HB) Model

# DCatch HB Model

Abstract a set of Happens Before rules.

$o_1 \overset{R}{\Rightarrow} o_2$

rule $R$ represents one type of causality relationship between a pair of operation

- This relation is transitive
    - if $o_1 \Rightarrow o_2$ and $o_2 \Rightarrow o_3$ then $o_1 \Rightarrow o_3$
- if neither $o_1 \Rightarrow o_2$ nor $o_2 \Rightarrow o_1$ holds, they are concurrent.

# Inter-node cuncurrency and communication

Nodes communicate with each other through message

Synchronus RPC(remote procedure call)

- node $n_1$ call PRC function f implemented by node $n_2$
- Rule-$M^{rpc}$ : $Create(r, n_1) \overset{M^{rpc}}{\Rightarrow} Begin(r, n_2)$;
  Rule-$M^{rpc}$ : $End(r, n_2) \overset{M^{rpc}}{\Rightarrow} Join(r, n_1)$

Asynchronous Socket

- node $n_1$ sends a message $m$ to node $n_2$
- Rule-$Msoc$ : $Send(m, n_1) \overset{M^{soc}}{\Rightarrow} Recv(m, n_2)$

## Custom Push-Based Synchronization Protocol

- Node $n_1$ updates a status $s$ to a dedicated corrdination node $n_c$, and $n_c$ notifies all subscribed nodes $n_2$ , about this update.
- Rule-$M^{push}$ : $Update(s, n_1) \Rightarrow Pushed(s, n_2)$

Decompose Rule-$M^{push}$ into three chains of causality relationship

- $Update(s, n_1) \Rightarrow Recv(s, n_c)$
- $Recv(s, n_c) \Rightarrow Send(s, n_c)$
- $Send(s, n_c) \Rightarrow Pushed(s, n_2)$

## Custom Pull-Based Synchronization Protocol

- node $n_2$ keeps polling $n_1$, about status $s$ in node $n_1$
- Rule-$M^{pull}$ : $Update(s, n_1) \overset{M^{pull}}{\Rightarrow} Pulled(s, n_2)$

# Intra-node cuncurrencynd communication

Synchronus multi-threaded cuncurrency

- classic fork/join causality
- the creation of thread $t$ happens before the execution of $t$ starts
- Rule-$T^{fork}$ : $Create(t) \overset{T^{fork}}{\Rightarrow} Begin(t)$
- the end of thread $t$ happens before the join of $t$
- Rule-$T^{join}$ : $End(t) \overset{T^{fork}}{\Rightarrow} Join(t)$

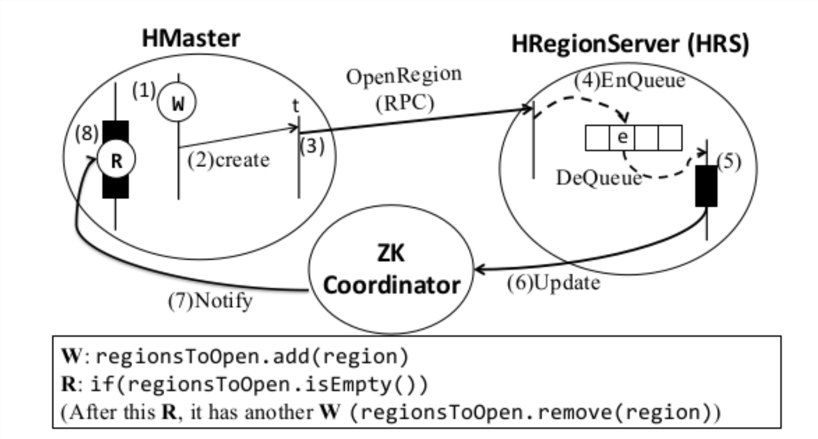
Asynchronous event-driven concurrency

- event is created before begin
- Rule-$E^{enq}$ : $Create(e) \overset{E^{enq}}{\Rightarrow} Begin(e)$

## Sequential program ordering

- Rule-$P^{req}$ : $o_1 \overset{P^{req}}{\Rightarrow} o_2$ if $o_1$ occures before $o_2$ during the execution of a regular thread
- Rule-$P^{nreq}$ : $o_1 \overset{P^{nreq}}{\Rightarrow} o_2$ if $o_1$ occures before $o_2$ during the execution of an event handler,a message handler, or an RPC function

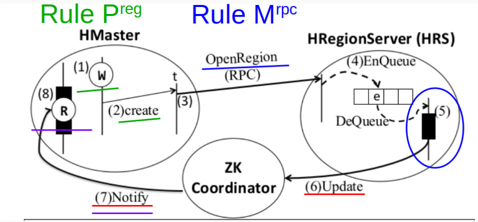
# Example with No HB graph



**W**: `regionsToOpen.add(region)`
**R**: `if(regionsToOpen.isEmpty())`
(After this **R**, it has another **W** `(regionsToOpen.remove(region))`

- To understand the timing between R and W
- Use HB-Graph

# Example with HB graph



Rule $P^{reg}$

Rule $M^{rpc}$

Rule $P^{nreg}$

Rule $M^{push}$

$$W \overset{P^{preg}}{\Rightarrow} Create(t) \overset{T^{fork}}{\Rightarrow} Begin(t) \overset{P^{reg}}{\Rightarrow}$$
$$Create(OpenRegion, HMaster) \overset{M^{rpc}}{\Rightarrow} Begin(OpenRegion, HRS) \overset{P^{nreg}}{\Rightarrow}$$
$$Create(e) \overset{E^{enq}}{\Rightarrow} Begin(e) \overset{P^{nreg}}{\Rightarrow} Update(RS\ldots OPENED, HRS) \overset{M^{push}}{\Rightarrow}$$
$$Pushed(RS\ldots OPENED, HMaster) \overset{P^{nreg}}{\Rightarrow} R$$

# DCatch tracing and trace analysis

# DCatch tracing and trace analysis

Given Happens-Before Model, build the DCatch tool

1. Trace the necessary operations
2. Build the Happens-Before graphs and perform analysis on top

# DCatch Tracing

DCatch produces a trace file at run time.

DCatch execute following tracing

- Memory-accesse tracing
    - Exhaustive approach is too expensive
    - Not trace all access
        - DCbugs are triggered by inter-node interaction, not every where in the software
- HB-related operation tracing
- Other tracing
    - trace lock/unlock

# HB-graph construction

HB-Graph

- DAG
- vertex $v$ represents an operation $o(v)$ recorded in DCatch trace
    - include memory access and HB-rule opration
- edge $e$ $v_1 \xrightarrow{e} v_2$ represents $v_1$ happens before $v_2$

How to construct HB-Graph?

1. Execute application and generate trace file
2. From trace file, make vertex
3. Add edges following MTEP rules

# DCbug candidate report

HB-Graph is huge

- $10^3 \sim 10^6$ nodes
- Naively analysis is too slow

To speed up analysis

- use the algorithm proposed by previous asynchronous race detection work
- Effective Race Detection for Event-Driven Programs[OOPSLA '13]

# Staitc pruning

# Static pruning

- Not all DCbug candidates can cause failures
- Avoid excessive false positive
  - treat certain instructions in software as failure instruction
  - failure instruction represent the occurrence of severe failure

To avoid excessive false positive...

- DCatch see if DCbug candidate impact towards the execution of any failure instruction
- if DCatch fails to find any failure impact for DCbug candidate, this DCbug candidate will be pruned out from the DCatch bug list

# DCbug triggering and validation

# DCbug triggering and validation

DCatch bug report still may not be harmful. Because...

- Custom synchronization undefined by DCatch
- The concurrent execution may not lead to any failure

To reliably expose truly harmful DCbugs, build end-to-end analysis-to-testing tool.

- an infrastructure that enable easy timing manipulation in distributed systems
- an analysis tool that suggests how to use the infrastructure to trigger a DCbug candidate

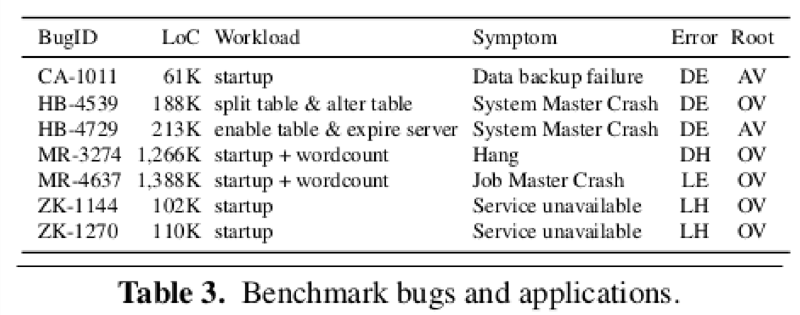
For DCbug candidate $(s, t)$, this tool execute

- $s \to t$
- $t \to s$

# Evaluation

# Evaluation

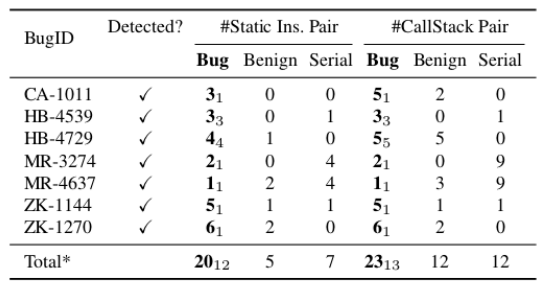
- Benchmarks
  - Cassanda
  - HBase
  - Hadoop
  - ZooKeeper
- Machine
  - Run each node of a distributed system in one virtual machine(M1)
    - A bug require twi physical machine (M1 & M2)
  - Ubuntu14.04
  - JVM1.7
  - M1 : Xeon CPU E5-2620
  - M2 : Core i7-3770
  - 64GB



| BugID | LoC | Workload | Symptom | Error | Root |
|---|---|---|---|---|---|
| CA-1011 | 61K | startup | Data backup failure | DE | AV |
| HB-4539 | 188K | split table & alter table | System Master Crash | DE | OV |
| HB-4729 | 213K | enable table & expire server | System Master Crash | DE | AV |
| MR-3274 | 1,266K | startup + wordcount | Hang | DH | OV |
| MR-4637 | 1,388K | startup + wordcount | Job Master Crash | LE | OV |
| ZK-1144 | 102K | startup | Service unavailable | LH | OV |
| ZK-1270 | 110K | startup | Service unavailable | LH | OV |

**Table 3.** Benchmark bugs and applications.

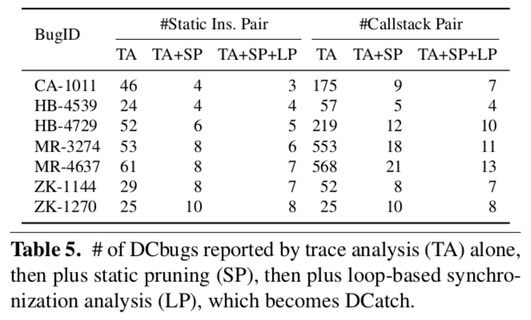
# Bug detection result



| BugID | Detected? | #Static Ins. Pair | | | #CallStack Pair | | |
|---|---|---|---|---|---|---|---|
| | | **Bug** | Benign | Serial | **Bug** | Benign | Serial |
| CA-1011 | ✓ | $\mathbf{3}_1$ | 0 | 0 | $\mathbf{5}_1$ | 2 | 0 |
| HB-4539 | ✓ | $\mathbf{3}_3$ | 0 | 1 | $\mathbf{3}_3$ | 0 | 1 |
| HB-4729 | ✓ | $\mathbf{4}_4$ | 1 | 0 | $\mathbf{5}_5$ | 5 | 0 |
| MR-3274 | ✓ | $\mathbf{2}_1$ | 0 | 4 | $\mathbf{2}_1$ | 0 | 9 |
| MR-4637 | ✓ | $\mathbf{1}_1$ | 2 | 4 | $\mathbf{1}_1$ | 3 | 9 |
| ZK-1144 | ✓ | $\mathbf{5}_1$ | 1 | 1 | $\mathbf{5}_1$ | 1 | 1 |
| ZK-1270 | ✓ | $\mathbf{6}_1$ | 2 | 0 | $\mathbf{6}_1$ | 2 | 0 |
| Total* | | $\mathbf{20}_{12}$ | 5 | 7 | $\mathbf{23}_{13}$ | 12 | 12 |

- DCatch has successfully detected DCbugs for all benchmarks
- 5/32 is Benign bug report
- For 7/32, DCatch mistakenly reports
    - some of them are unidentified RPC function

# false-positive pruning



| BugID | #Static Ins. Pair | | | #Callstack Pair | | |
|---|---|---|---|---|---|---|
| | TA | TA+SP | TA+SP+LP | TA | TA+SP | TA+SP+LP |
| CA-1011 | 46 | 4 | 3 | 175 | 9 | 7 |
| HB-4539 | 24 | 4 | 4 | 57 | 5 | 4 |
| HB-4729 | 52 | 6 | 5 | 219 | 12 | 10 |
| MR-3274 | 53 | 8 | 6 | 553 | 18 | 11 |
| MR-4637 | 61 | 8 | 7 | 568 | 21 | 13 |
| ZK-1144 | 29 | 8 | 7 | 52 | 8 | 7 |
| ZK-1270 | 25 | 10 | 8 | 25 | 10 | 8 |

**Table 5.** # of DCbugs reported by trace analysis (TA) alone, then plus static pruning (SP), then plus loop-based synchronization analysis (LP), which becomes DCatch.

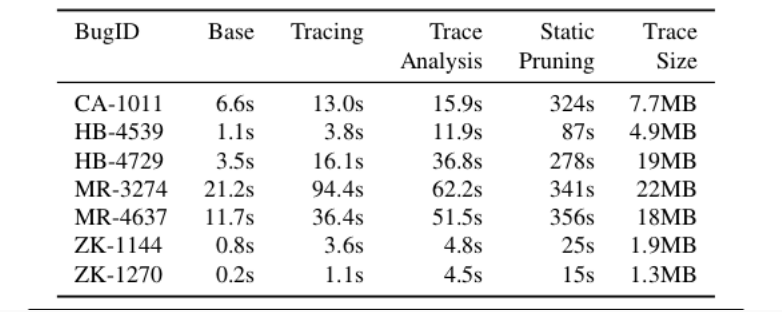
- Static pruning pruned out a big portion of DCbug candidates
- loop-based synchronization is effective

# False negative discussion

DCatch could miss DCbugs for several reason

- Because of the configure of static pruning, DCatch miss silent bug
- DCatch miss the DCbug between communication-related memory accesses and communication-unrelated access
- DCatch may not process extreamly large traces

# Performance



| BugID | Base | Tracing | Trace Analysis | Static Pruning | Trace Size |
|---|---|---|---|---|---|
| CA-1011 | 6.6s | 13.0s | 15.9s | 324s | 7.7MB |
| HB-4539 | 1.1s | 3.8s | 11.9s | 87s | 4.9MB |
| HB-4729 | 3.5s | 16.1s | 36.8s | 278s | 19MB |
| MR-3274 | 21.2s | 94.4s | 62.2s | 341s | 22MB |
| MR-4637 | 11.7s | 36.4s | 51.5s | 356s | 18MB |
| ZK-1144 | 0.8s | 3.6s | 4.8s | 25s | 1.9MB |
| ZK-1270 | 0.2s | 1.1s | 4.5s | 15s | 1.3MB |

- DCatch tracing causes 1.9x $\sim$ 5.5x slowdown
- Static pruning is the most time consuming phase

# Conclusion

# Conclusion

- Designed automated DCbug detection tool for large real-world distributed system
- DCatch HB model combine causaly relationship in single machine system and distributed system
- DCatch is just a starting point in combating DCbugs